디자인 패턴

💡 ***디자인 패턴(Design Pattern)***은 애플리케이션 개발에서 반복되는 문제들에 대한 동일한 해결책으로, 비즈니스 요구사항 처리 시 만들어진 해결책 중 모범 사례들에 해당된다.

프로그래밍 시 당면하게 되는 유사한 문제들에 대한 일종의 ***요리 레시피***인 셈. 표준화된 레시피를 이용해 요리를 하되, 개인의 입맛에 맞춰 레시피에 조금씩 이것 저것 추가하는 방식의 개발 과정.

디자인 패턴 장점

- **재사용성** : 반복적인 문제들에 대한 해결책이기 때문에 유사한 상황에 대해 쉽게 적용할 수 있다.
- **가독성** : 구조가 일정하고 명확하게 작성되어 있기 때문에 코드 이해 및 유지보수가 쉽다.
- **유지 보수성** : 코드를 모듈화하기 쉬우며, 변경이 필요한 경우 해당 모듈만 수정하기 때문에 유지 보수가 쉽다.
- **확장성** : 새로운 기능 추가 및 변경 시 기존 코드를 변경하지 않고 새로운 기능의 확장이 가능하다.
- **안정성 & 신뢰성** : 검증된 솔루션을 제공한다.

디자인 패턴 종류 - GoF 디자인 패턴

수 많은 디자인 패턴들 중 소프트웨어 공학에서 가장 많이 사용되는 분류 방식으로 ***GoF(Gang of Four) 디자인 패턴***이 있다. 목적에 따라 크게 **생성(Creational)**, **구조(Structural)**, **행위(Behavioral)**의 3가지로 나뉜다.

- **생성 패턴** : 객체 생성 시 사용되는 패턴으로, 객체 수정해도 호출부는 영향을 받지 않게 한다.
- **구조 패턴** : 객체를 조합해 더 큰 구조를 만드는 과정 즉, 연결에 관한 패턴이다.
- **행위 패턴** : 객체 간의 알고리즘 및 책임을 분배하며 협력하는 관계에 대한 패턴이다.

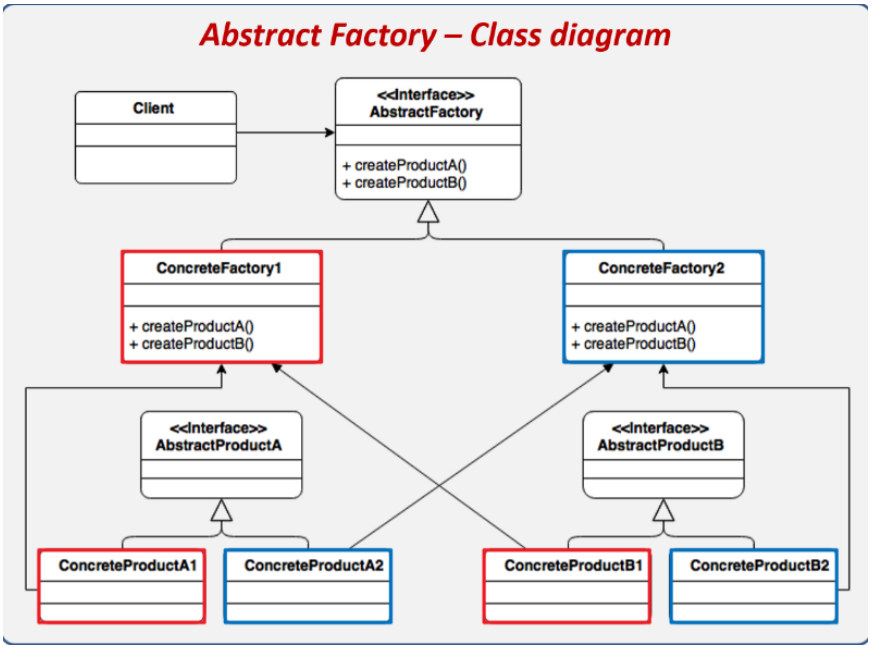
GoF 디자인 패턴 분류⬇️

생성(Creational)	구조(Structural)	행위(Behavioral)
추상 팩토리(Abstract Factory)	어댑터(Adaptor)	책임 연쇄 (Chain of Responsibility)
빌더(Builder)	브리지(Bridge)	커맨드(Command)
팩토리 메서드(Factory Method)	컴포지트(Composite)	인터프리터(Interpreter)
프로토 타입(Prototype)	데코레이터(Decorator)	이터레이터(Iterator)
싱글톤(Singleton)	퍼사드(Facade)	미디에이터(Mediator)
	플라이웨이트(Flyweight)	메멘토(Memento)
	프록시(Proxy)	옵저버(Observer)
		스테이트(State)
		스트레티지(Strategy)
		템플릿 메서드 (Template Method)
		비지터(Visitor)

생성 패턴(Creational Pattern)

객체의 생성과 관련된 패턴으로 객체 생성 및 참조 과정을 캡슐화 하여 객체 생성 및 수정에도 프로그램 구조에 영향을 받지 않게 담당한다. 쉬운 이해를 위해 필자는 **자동차 생산 공장**에 빗대어 해석했다.

- **추상 팩토리(Abstract Factory)** : 구체적인 클래스를 지정하지 않고 상황에 맞는 객체를 생성하기 위해 인터페이스만 제공하는 패턴



→ ex) 자동차 생성에 `바퀴`, `유리`, `엔진`이 필수로 들어가야 한다. 벤츠 공장의 바퀴, BMW의 바퀴 모두 "바퀴"를 만들기 때문에 `바퀴 인터페이스`를 구현한 `BMW바퀴`, `벤츠바퀴`를 각각 정의한다. 엔진도 마찬가지로 `엔진 인터페이스`를 구현한 `BMW엔진`, `벤츠엔진` 클래스를 각각 정의한다. → `벤츠바퀴`, `벤츠엔진`은 `벤츠` 라는 공통점을 가지기 때문에 하나로 묶을 수 있다. `BMW` 또한 마찬가지. 이처럼 공통된 팩토리들을 모아둔 것이 추상 팩토리이다.

- **빌더(Builder)** : 객체의 생성과 표현을 분리해 객체를 단계적으로 생성하는 패턴

  → ex) `자동차`를 단계 별로 부품을 하나씩 추가하면서 차를 완성하는 과정

- **팩토리 메서드(Factory Method)** : 객체 생성을 서브 클래스로 분리해서 위임하는 패턴

  → ex) 차를 만드는 공통적인 공정(메서드)을 갖는 `자동차 공장`(인터페이스)만 제공함. 지역에 따라 각 공장 별 생성 자동차의 종류가 다름. `경기도 공장`은 SUV, `충남 공장`은 스포츠카 등등... → 클라이언트는 팩토리 메서드를 통해 원하는 차종을 생성할 수 있음.

- **프로토타입(Prototype)** : 객체를 복사하여 새로운 객체를 생성하는 패턴 → 기존 객체가 템플릿이 됨.

  → ex) `스포츠카`를 하나 만들어둔 뒤, 이걸 그대로 복사해서 똑같은 차를 만드는 방식

- **싱글톤(Singleton)** : 한 클래스마다 인스턴스를 단 하나만 생성해서 이걸 보장해주는 셈. 어느 곳에서도 접근할 수 있게 제공하는 패턴

  → ex) `차`를 매 번 새롭게 생성해주는 것이 아닌, `차`를 딱 한 대만 만들어 둔 뒤, 누구에게나 그 차를 빌려주는 방식.

구조 패턴(Structural Pattern)

객체들의 구성과 연결에 대한 패턴이다. 객체들의 협력 및 시스템의 유연한 확장을 돕는 패턴이다.

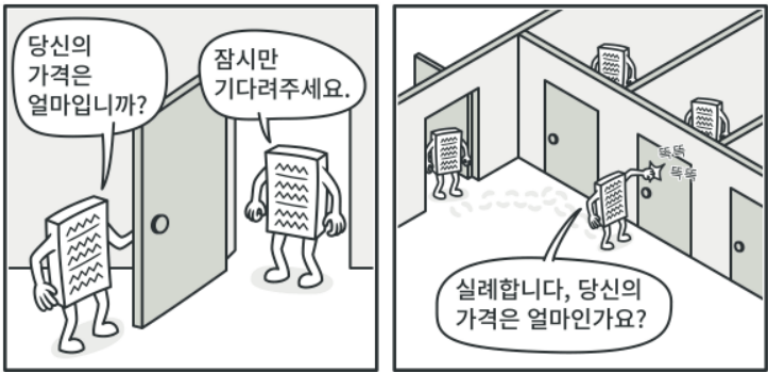
- **어댑터(Adaptor)** : 다른 방식으로 동작하는 인터페이스를 우리가 의도하는 인터페이스로 변환

  → ex) 일본의 110V는 한국과 맞지 않지만 어댑터를 이용하면 220V도 지원된다.

- **브리지(Bridge)** : 추상화와 구현을 분리해서 연결하는 패턴이다.

  → ex) 자동차 공장에서 차의 모델(추상화)와 차를 만드는 방식(구현)을 분리하고 연결하는 방식이다. `CarModel` 은 (SUV, 세단) 등을 의미하고, 이걸 실제로 만드는 방식은 `CarFactory` 인터페이스를 구현하여 생산하는 방식이다.

- **컴포지트(Composite)** : 개별 객체랑 여러 개별 객체가 모여 있는 복합 객체를 동일하게 다루어 트리 구조의 객체를 구성하는 패턴이다.



  → ex) `상자` 안에 있는 물건에 대해 질문한다고 가정하자. `상자` 안에는 `상자` 또는 `제품`이 있을 수 있다. `상자` 안의 물건에 대해 물어봤을 때 안에 `작은 상자`가 있었다면 `작은 상자`에 대해 똑같은 방식으로 안에 무엇이 있는지 질문할 수 있다. 만약 `제품`이면 `제품`에 대

해 이야기 해주는 것이다. 이처럼 계층 구조로 하위 계층까지 수행할 작업을 전달하는 과정에 해당된다.

- **데코레이터(Decorator)** : 객체들의 결합을 통해 기능을 동적으로 유연하게 확장할 수 있도록 하는 패턴이다. 상속 없이 객체를 감싸는 방식을 사용한다.

  → ex) `커피`가 있으면 그 `커피`에 `시럽`을 추가할 수도, `우유`를 추가할 수도 있다.

- **퍼사드(Facade)** : 서브 시스템을 더 쉽게 사용할 수 있도록 단순한 인터페이스를 제공하는 패턴

  → ex) 전화 주문을 위해 매장에 전화를 건다면, 전화를 받는 `상담원`이 퍼사드에 해당된다. 상담원은 `주문`, `포장`, `배달`, `결제 처리` 등 복잡한 서브 시스템을 간단히 사용할 수 있게 돕는 `음성 인터페이스`를 제공하는 셈.

- **플라이웨이트(Flyweight)** : 특정 클래스의 인스턴스 한 개를 가지고 여러 개의 '가상 인스턴스'를 제공할 때 사용하는 패턴. 여러 객체의 공통된 부분을 공유해서 메모리 사용을 최적화 하는 패턴이다.

  → ex) 컴퓨터 게임 내에서 `총알의 색`, 게임 내 `배경화면`, `총알 모양` 등은 사용자에 따라 달라지지 않는 동일한 값이며, 이걸 매 사용자 마다 생성하지 않고 해당 값들이 저장되어 있는 객체 하나를 공유하는 방식이 플라이 웨이트이다. 총알의 위치, 쏜 방향 등 외부에서 변경되는 객체의 나머지 상태들을 더 많이 저장할 수 있다.

- **프록시(Proxy)** : 특정 객체를 직접 참조하지 않고 해당 객체를 대행하는 객체를 통해 접근하는 방식. 원래 객체에 대한 접근을 제어하기 때문에 원래 객체에 전달되기 전, 후에 추가적인 작업 수행이 가능하다.

  → ex) 영화를 직접 재생하는 방식이 아닌, `영화 스트리밍 서비스`를 이용한 영화 재생. 이렇게 될 경우 영화 상영 전에 광고를 보여주거나 접속 속도 체크 및 지연 관리 등의 추가 작업이 가능하다.

행위 패턴(Behavior Pattern)

객체 및 클래스 간의 상호작용, 즉 행동에 대해 다루는 디자인 패턴이다. 객체 간의 책임의 분배, 상호 작용 방식이나 상태 변화 등을 정의하여 시스템의 유연성과 확장성을 높인다.

- **책임 연쇄(Chain of Responsibility)** : 요청에 대한 처리 객체를 집합으로 만들고, 각 핸들러들이 요청을 전달하는 과정에서 처리할지, 넘길지에 대해 결정하는 방식이다.

  → ex) `상담원`이 고객 요청을 받고 해결하거나 넘김 → `팀 리더`가 `상담원`이 넘긴 복잡한 문제를 해결하거나 처리할 수 없는 요청은 또다시 넘김 → `매니저`가 최종 해결을 맡음

- **커맨드(Command)** : 요청에 대한 정보가 포함된 객체로 변환하는 패턴이다. 클라이언트는 명령만 보내면 되는 셈.

  → ex) 리모콘을 이용해 TV를 키고 끌 때 사용자는 버튼을 누르기만 해도 기능을 실행할 수 있으며, 기능 뒤에 어떤 메서드들이 작동하는지에 대해 알 필요가 없다.

- **인터프리터(Interpreter)** : 주어진 언어문법을 위한 표현 수단을 정의하고, 해당 언어로 구성된 문장을 해석하는 패턴이다.

  → ex) 영어의 한국어 번역 시, 영어의 문장 구조와 단어의 의미 등의 규칙에 따라서 영어를 해석하고, 이를 한국어로 번역함.

- **이터레이터(Iterator)** : 내부 구조를 노출하지 않으면서 해당 객체의 집합 원소에 순차적으로 접근하는 방식. 컬렉션의 순회 동작을 반복자라는 별도의 객체로 추출하는 것이 핵심이다.

  → ex) 한국의 주요 유적지들을 순회하고 싶은데 이들을 일일이 찾아다닐 수 없는 셈. `네비게이션`이 안내하는 대로, 혹은 `관광 가이드`가 데려가는 대로 이동하는 방식이 반복자에 해당된다.

- **미디에이터(Mediator)** : 한 집합 내 객체 간의 상호작용을 캡슐화 하는 패턴이다. 객체 간의 직접 통신을 제한하고 중재자 객체를 통해서만 협력하도록 하는 방식이다.

  → ex) 공항 관제 구역으로 들어오는 다양한 비행기들은 각 비행기들과 직접 연락하지 않고 관제탑을 통해서만 연락한 뒤 이착륙한다. 또한 관제탑은 특정 지역에서만 비행기들의 이착륙을 관리한다.

- **메멘토(Memento)** : 객체의 상태 정보를 저장하고 필요에 따라 상태를 복원하는 방식이다. 객체의 세부 사항은 공개하지 않으면서 원하는 상태에 대해 저장 및 복원할 수 있게 돕는다.

  → ex) '`실행 취소`' 기능은 이전 자신의 상태에 대한 스냅샷의 일부 메타 데이터를 갖지만 원래 객체의 상태는 갖지 않는다. '`실행 취소`'를 작동시킬 경우 가장 최근의 메타 데이터

를 저장하는 `메멘토`를 가져와서 본체인 `텍스트 편집기`에게 해당 상태로의 롤백을 요청하면 `편집기`는 자신의 상태를 변경한다.

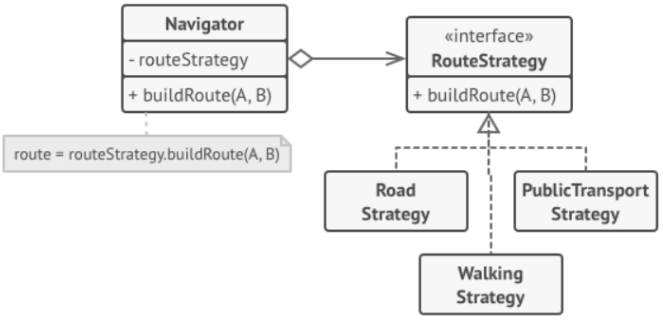
- **옵저버(Observer)** : 객체의 상태 변화를 관찰하여 상태가 변하거나 이벤트가 발생할 때마다 알리는 메커니즘이다.

  → ex) `출판사`들은 그들을 구독하는 `구독자`들을 갖는다. `구독자`들은 모두 동일한 인터페이스를 구현하고 `출판사`는 해당 인터페이스를 통해 `구독자`들의 리스트를 참조해 그들에게 알림을 전송한다.

- **스테이트(State)** : 객체의 내부 상태가 변경될 때, 해당 상태에 따라 객체가 행동을 변경하는 패턴이다. 객체의 모든 가능한 상태에 대해 클래스를 만들고, 상태 별 행동들을 해당 클래스로 추출한다.

  → ex) 스마트폰의 잠금 해제 시 → 다양한 어플리케이션 실행 가능 / 스마트폰 잠금 시 → 어떤 버튼을 눌러도 잠금 해제 관련된 화면만 뜸

- **스트래티지(Strategy)** : 행동을 클래스로 캡슐화 해서 동적으로 행동을 바꿀 수 있도록 하는 패턴. 특정 작업을 다양한 방식으로 수행하는 여러 클래스들을 전략이라는 별도의 클래스에 넣어두는 셈.



- **템플릿 메서드(Template Method)** : 부모 클래스에서 알고리즘의 골격을 정의해주지만, 자식 클래스에서 전체 수행 구조는 바꾸지 않으면서 특정 단계만 오버라이드해서 사용하는 디자인 패턴이다.

  → ex) 주택 건설 시 전체적인 공사 방식은 유사하지만 클라이언트의 요구 사항에 의해 내부 인테리어나 건축 단계가 조금씩 변경이 될 수 있다.

- **비지터(Visitor)** : 로직을 가지고 있는 객체인 방문자(Visitor)가 로직을 적용할 객체를 방문하여 실행하는 패턴이다.

  → ex) 동물원의 동물들을 돌아가면서 마주할 때 호랑이에게는 고기를, 기린에게는 풀을, 원숭이에게는 바나나를 주는 방식에 해당된다. 방문자에게 전달된 인수를 통해 어떤 메서드가 더 적합한지를 알려주는 방식이다.